TU Clausthal

Clausthal University of Technology

Scalable Multi-Agent Simulation based on MapReduce

Tobias Ahlbrecht, Jürgen Dix and Niklas Fiekas

IfI Technical Report Series IfI-16-03

IfI

Department of Informatics
Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Tobias Ahlbrecht
Contact: tobias.ahlbrecht@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

Scalable Multi-Agent Simulation based on MapReduce

Tobias Ahlbrecht Jürgen Dix Niklas Fiekas

Department of Informatics
Clausthal University of Technology
Julius-Albert-Str. 4
D-38678 Clausthal-Zellerfeld, Germany

Abstract

Jason is perhaps the most advanced multi-agent programming language based on *AgentSpeak*. Unfortunately, its current Java-based implementation does not scale up and is seriously limited for simulating systems of hundreds of thousands of agents.

We are presenting a scalable simulation platform for running huge numbers of agents in a Jason style simulation framework. Our idea is (1) to identify independent parts of the simulation in order to parallelize as much as possible, and (2) to use and apply existing technology for parallel processing of large datasets (e.g. MapReduce).

We evaluate our approach on an early benchmark and show that it scales up linearly (in the number of agents).

1 Introduction

The work reported in this paper is part of a bigger project on using agent-based simulation for quality control of software development processes [1]. In this project we need a platform that is able to simulate a huge number of agents, (hundreds of thousands or even more).

Current approaches implemented in Java often do not scale up (see [2] for a detailed discussion). Similarly, declarative approaches (e.g. those based on *AgentSpeak*) are well suited for modeling simulations, but do not support efficient implementation.

Here we focus on a new approach for implementing scalable multi-agent simulation platforms with MapReduce. The main idea is to identify parts of the simulated environment that are completely independent from each other and can thus be processed in parallel. This is particularly useful in scenarios based on large existing datasets, but can also be applied to multi-agent simulation in general.

In the following we give a very brief introduction to Jason and MapReduce and comment on related work. The main part is Section 2, where we show how Jason can be interpreted in a way that is compatible with MapReduce. While previous approaches have used limited agent models [8] or restricted languages [11] our approach supports full Jason-style *AgentSpeak*. We believe that similar agent languages can be translated accordingly.

Key points of any simulation are (1) modeling and (2) implementing the environment: we elaborate on both in Section 3. Finally we evaluate our approach in Section 4 using a benchmark for our early proof of concept implementation[1] and conclude with Section 5.

1.1 Jason

Jason is a Java based platform for multi-agent simulation with an extended version of *AgentSpeak* [4]. *AgentSpeak* is a language to describe BDI agents that mixes a declarative approach to reasoning (Prolog) and an imperative way of stating plans [9]. Jason extends the language with useful functionality such as agent communication. Jason is widely used [3] but does not scale well when the simulation size is increased beyond thousands of agents, even when the agents are very simple.

1.2 MapReduce

MapReduce is a programming paradigm designed to simplify the parallel processing of large datasets [5] by abstracting away low level architecture (single thread, multi-core computer, grid of commodity computers), synchronization, error recovery, locking and distribution of work among the nodes of a cluster. The algorithm is defined in terms of *map* and *reduce* functions that operate on key value pairs. Map functions operate independently on key value pairs $\langle k, v \rangle$. After a shuffling step that groups items by their keys, *reduce* functions operate on sequences of values in each group:

$$\text{Map} : (K, V) \to (K, V)^* \ \ ; \ \ \text{Reduce} : (K, V^*) \to (K, V)^*$$

Algorithms in terms of these functions can be executed using a MapReduce framework like Spark[2], Hadoop[3], MR4C[4], MapReduce-MPI[5] or Disco[6], which automatically partition the dataset for parallel execution.

[1] Source code available at `https://github.com/niklasf/pyson`
[2] `http://spark.apache.org/`
[3] `http://hadoop.apache.org/`
[4] `https://github.com/google/mr4c`
[5] `http://mapreduce.sandia.gov/`
[6] `http://discoproject.org/`

1.3 Related Work

There are several design patterns for MapReduce that have been used outside of agent simulation. Lin and Schatz [7] describe algorithms that allow communication along the edges of graphs. Zhang et al. [14] provide a technique for parallelizing spatial joins. These have then been used in agent system simulation with agent models that have been restricted accordingly: Radenski [8] uses graph algorithms to simulate cellular automatons. Wang et al. [11] use spatial joins for behavioral simulations, where agent actions are restricted to associative operations on the environment.

2 Translating Jason to MapReduce

When agents deliberate but do not communicate or execute actions in the environment they can be executed independently in Map steps. In this section we discuss key requirements for a Jason interpreter that allows doing that. The key point is to represent the state of agents and the state of the environment in *key value pairs* such that *actions that advance the simulation can be performed efficiently with Map and Reduce steps*.

Most MapReduce platforms commit datasets to disk after each MapReduce step. However this overhead can be avoided for multi-agent simulation: In case of data loss computation steps can simply be repeated. We therefore choose Apache Spark as our underlying platform. Spark features the concept of *Resilient Distributed Datasets* (RDDs) with configurable levels or persistence. Additionally, Spark uses the scripting language Python as one of the primary supported languages. This allows us to use Python as a single language for the platform as well as for scripting the simulated environment and available actions. There are three key requirements for the Jason interpreter:

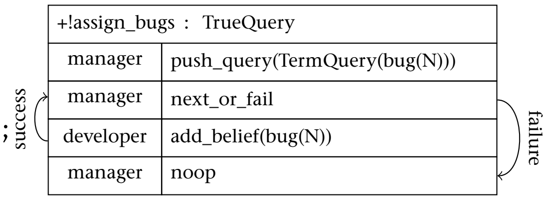
- **Serializability**: The state of agents must be serializable at any given time to allow Spark to serialize and transmit them to other nodes of the cluster.

- **Ability to pause and resume individual agents**: In distributed computing local operations are near-instant while network operations take orders of magnitudes more time. An agent waiting for data from the network needs to be paused in order not to block the execution of other agents.

- **Memory efficiency**: The interpreter must have a low memory footprint so that hundreds of thousands of agents can fit into main memory.

```
+!assign_bugs <-
  for (bug(N)) {
    .send(developer,
          tell, bug(N));
  }.
```

(a) Jason source code



(b) The constructed control flow graph

Figure 1: Example: A `manager` agent sends bug details to a `developer` agent

For memory efficiency we embed native Python data types directly into Jason (`bool`, `int` and `long` and `float` for numerics, `tuple` for lists). Variables and belief literals are defined as classes in Python (`Var()` and `Literal(functor, args)`). All other Python objects are treated as atoms. To avoid making copies of objects, all substitutions (mappings of variables to terms) are kept in a separate dictionary. Additionally, agents have a stack of substitutions and choice points that allows them to undo failed partial unifications.[7]

To allow pausing and resuming individual agents (even while they are executing a Prolog query) we use Python generators to iterate over alternatives, with a technique similar to YieldProlog[8]. Finally the Python implementation PyPy guarantees serializability of Python objects including functions, closures and generators.

For AgentSpeak(L) the control flow in a plan is linear. Jason defines additional control structures such as branches and loops. To capture both we represent plans as a control flow graph where nodes are high level instructions. Each node has at most two outgoing edges labeled `success` or `failure` that are followed depending on the result of the current instruction. If a node does not have the corresponding edge this is interpreted as plan achievement or plan failure respectively.

Intentions in *AgentSpeak* are defined as a stack of partially instantiated plans [9]. To avoid copying plans for each instantiation we use a separate intention data structure instead. The data structure contains (i) the instantiated plan head from the point of view of the caller, (ii) a pointer to the current instruction in the control flow graph, (iii) the current substitution `scope` (mapping of variables to terms), (iv) stacks to undo unifications and continue with a different choice (`stack`, `query_stack`, `choicepoint_stack`). The corresponding set of instructions is given in the appendix.

Observation 1 (Correct-, and Completeness) *The described interpreter satis-*

[7] This technique is well known in Prolog interpreters [12, 13].

[8] http://yieldprolog.sourceforge.net/

fies the hard requirements outlined above. In addition, all Jason programs can be transformed to programs in our instruction set.

3 Handling the Environment

To simulate the environment, a number of different object types have to be modeled. Possible *actions* and *percepts* make up a major part, as they imply the environment's behavior and thus determine the computational effort. Environments need a notion for each "thing" that is not an agent: we call it *artifact*.

The entire state of the simulation is stored in key value pairs. It comprises the agents $\langle uuid, agent \rangle$ and artifacts from the environment. A cycle of the simulation starts with a map step where each agent state is mapped to the next. Messages to other agents are emitted as key value pairs using a Jason-style belief annotation for the sender: $\langle recipientUuid, message[source(senderUuid)] \rangle$. Actions selected by the agent emit additional key values pairs (usually of the form $\langle affectedArtifactUuid, action \rangle$).

The actual effects of the actions are computed in a reduce phase where key value pairs are grouped by recipient or affected artifact. Reduce operations in Spark must be associative. Additionally commutativity is a reasonable requirement to achieve deterministic results even when the order of the values is non-deterministic. Actions that return results must include the UUID of the agent so that results can be emitted as a key value pair $\langle uuid, resultMessage \rangle$.

Values for distinct keys are reduced in parallel. This leads directly to the following observation.

Observation 2 *The environment needs to be designed such that potentially conflicting actions always affect the same key.*

While this can be trivially achieved by using a monolithic environment with a single key, it is likely that the reduction for that key will be a bottleneck. Thus, to allow parallel execution, we need the following complementary goal.

Observation 3 *Independent actions must affect distinct keys.*

For many scenarios there is a natural way to decompose the environment into key value pairs. For example [11] partition a spatial environment into overlapping areas to simulate social force. Since areas overlap, the same action (effects) may be sent to multiple keys. Summation is used as an associative and commutative reduce operation. However, as not all simulations decompose spatially (see the *Simulating Software Evolution* scenario) we propose the following additions:

- Instead of hardcoding the concept of spatial location we introduce groups that agents can subscribe to and send multicast messages to. This mechanism will also be exploited for percept generation and distribution.

- Deterministic reservoir sampling [10] as an associative and commutative operation to fairly select one of multiple conflicting actions. This works for arbitrary actions since they no longer have to be associative and/or commutative themselves.

Currently, the whole environment has to be hand-coded as a Python script. The next step is to provide a thin wrapper around Spark to abstract away from its concrete functionality so as not to burden the user with having to learn everything about MapReduce in order to use the platform. In a later step, the final environment metamodel will be combined with our already existing Jason metamodel to provide the user with schematic modeling facilities (i.e. diagramming) to enable kick-starting new projects.

3.1 Application: Simulating Software Evolution

As mentioned before, the platform is part of a bigger project on simulating software development processes using agent-based technology to gain insights on (specific) software evolution. In this scenario, agents can perform abstract modifications on the software project, i.e. “fix bugs” or “refactor methods”. Representing the developers with simple agents already proved a viable solution [6]. However, to get more detailed results, it is necessary to equip agents with better reasoning and planning capabilities. This will enable them to adopt goal-oriented behavior, e.g. based on code change patterns. Furthermore, beliefs will be crucial to simulate how the agents gain experience in the process (see [1]).

Exploiting MapReduce will also greatly benefit the simulations’ running time, enabling those of large software projects with many (behaviorally) complex developers and even those where multiple projects form an ecosystem exchanging resources and information.

4 Evaluation

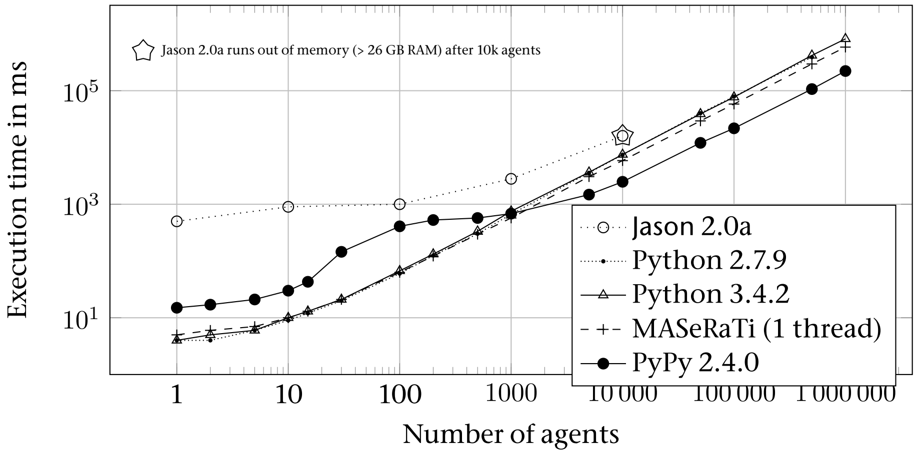


Figure 2: Execution times of the counting scenario for increasing numbers of agents

The authors of [2] have developed a simple benchmark to compare several platforms based on different implementations. It models the throughput of the interpreter on a single node (it relates to the implementation described in Section 2). We compare the performance of our platform running on different Python interpreters (Python 2, Python 3, PyPy) with the performance of other platforms (Jason, *Maserati*)[9], see Fig. 2.

Jason 2.0a runs out of memory for 50 000 agents, but could potentially complete the simulation on a machine with even more RAM. The other platforms scale roughly linearly as expected for this simple scenario. We achieve the best performance with PyPy which uses *Just-In-Time compilation* and *hotspot optimization* (see the disproportional speedup for a medium number of agents).

5 Conclusion

We have presented a scalable Jason interpreter that is part of a bigger project on quality control of software development processes (see [1]). However, we believe our approach is rather general and can be applied to similar agent languages based roughly on *AgentSpeak* (which allows us to use the built-in

[9]The test environment is a pristine Debian Jessie using an Intel Xeon CPU @ 4 x 2.30 GHz and 26 GB RAM.

modelling constructs). All that needs to be done is to find a suitable translation of this language into MapReduce (as described in Section 2). An advantage of our approach is the possibility to use off-the-shelf professional tools to deal with MapReduce.

Our evaluation shows linear scalability (in the number of agents) in a simple benchmark, even for a reimplementation of Jason. It remains to test other benchmarks and to tailor our system for the application in the planned project. But we are planning to apply our approach also to other areas, where parallelization in the simulation of an environment pays off.

References

[1] Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, Jens Grabowski, Verena Herbold, Daniel Honsel, Stephan Waack, and Marlon Welter. Agent-based simulation for software development processes. Technical Report IfI-16-02, TU Clausthal, September 2016. (to appear).

[2] Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas Philipp Kraus, and Jörg P. Müller. An architecture for scalable simulation of systems of cognitive agents. *International Journal of Agent-Oriented Software Engineering*, 2016.

[3] Rafael Bordini and Jürgen Dix. Chapter 13: Programming multi-agent systems. In Gerhard Weiss, editor, *Multiagent systems*, pages 587–639. MIT-Press, 2013.

[4] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[6] Verena Honsel, Daniel Honsel, Steffen Herbold, Jens Grabowski, and Stephan Waack. Mining software dependency networks for agent-based simulation of software evolution. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 102–108. IEEE, 2015.

[7] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in MapReduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG ’10, pages 78–85, New York, NY, USA, 2010. ACM.

[8] Atanas Radenski. Using MapReduce streaming for distributed life simulation on the cloud. ECAL, 284-291(2013), 2013.

[9] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-agent World: Agents Breaking Away*, MAAMAW '96, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[10] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[11] Guozhang Wang, Marcos Antonio Vaz Salles, Benjamin Sowell, Xun Wang, Tuan Cao, Alan J. Demers, Johannes Gehrke, and Walker M. White. Behavioral simulations in MapReduce. *CoRR*, abs/1005.3773, 2010.

[12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[13] Michael Winikoff. W-Prolog, 1996.

[14] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. SJMR: parallelizing spatial join with MapReduce on clusters. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*, pages 1–8, 2009.

A Set of Instructions

These instructions are used as an intermediate representation of Jason programs:

noop(agent, intention) Does nothing and succeeds always.

add_belief(term, agent, intention) Applies the current substitution to `term` and adds it to the belief base. Triggers a belief addition event.

remove_belief(term, agent, intention) Unifies `term` with the first matching belief and removes it from the belief base. Triggers a belief removal event.

test_belief(term, agent, intention) Tries to find a substitution such that `term` is a logical consequence of the belief base. Triggers a belief test event.

call(trigger, goal_type, term, agent, intention) Tries to find a plan matching `trigger`, `goal_type` and `term` and adds it as a subplan to the current intention.

call_delayed(trigger, goal_type, term, agent, intention) Tries to find a plan matching `trigger`, `goal_type` and `term` and creates a new intention with it.

push_query(query, agent, intention) Starts the Prolog query `query` and adds the resulting Python generator to the query stack. This is also used for actions that can yield multiple results.

next_or_fail(agent, intention) Tries to advance the topmost generator.

pop_query(agent, intention) Removes the topmost generator from the stack.